The Virtual World Framework: Implementing a Web Based Client Side Simulator

Robert Chadwick
Katmai, in support of the Advanced Distributed Learning Initiative
Orlando, FL
robert.chadwick.ctr@adlnet.gov

David Easter
Lockheed Martin
Cary, NC
david.easter@lmco.com

ABSTRACT

The Virtual World Framework (VWF) is a browser-based collaborative simulation system designed with the goal of creating content in a shared immersive space. For the Modeling & Simulation community, the VWF represents an evolution of the traditional online learning model by allowing the integration of three dimensional environments with other, more typical, content. Traditionally, offering learners an experience in such an environment required the instructional designer to craft content in some external system, often requiring specialty plug-ins and equipment. Many of these systems (i.e. Second Life, or OpenSim) require authors to learn new programming languages and methods that are not typically taught to instructional designers. By leveraging the web as a platform, the VWF makes it possible for a much wider range of authors to create immersive environments. Additionally, the VWF makes deploying this content to the end user much simpler than traditional immersive training packages. While other simulation environments require desktop software installation or browser plugins, the VWF uses cutting edge HTML5 capabilities to deliver its content to a variety of devices like desktops, tablets, and even phones, without any effort by the end user. Users simply type in a URL as they would for any website, and they can access the environment seamlessly. This has powerful benefits; including allowing immersive experiences on computers with strict security that prohibits many traditional solutions. Finally, the VWF architecture makes hosting a server simple. The VWF builds the simulated environment on each client, allowing the server to manage only synchronization. This allows a server to host many users while utilizing little bandwidth and computational power. This paper will describe the VWF architecture in detail and explain how the design decisions support the goal of wider access to immersive simulation.

ABOUT THE AUTHORS

Rob Chadwick is a 3D programmer and artist who has been working in the 3D content field for more than ten years. In that time, he has designed custom game engines, managed render farms, and written productivity tools. He has taught entry level video and photo editing, and offered courses on advanced 3D animation. He has designed simulations for the FBI and the US Navy using custom and off-the-shelf game software, and animated virtual spaces for architecture and interior design companies. While primarily interested in rendering technologies, Rob has worked for the last year with Advance Distributed Learning to foster interoperability of 3D assets. He holds a Bachelors of Computer Science from Virginia Tech.

David Easter has been creating graphics simulations on emerging platforms for more than 20 years. As a founding member of Virtus Corporation, he co-authored the award-winning Virtus Walkthrough in an era before GPUs. Later, as Chief Architect at 3Dsolve, he developed the engine underlying U. S. Army TRADOC's first Level 4 Interactive Multimedia Instruction. More recently at Lockheed Martin, and working with the office of the Under Secretary of Defense for Personnel and Readiness, he is bringing simulation training to the web as architect of the Virtual World Framework.

The Virtual World Framework: Implementing a Web Based Client Side Simulator

Robert Chadwick
Katmai, in support of the Advanced Distributed Learning Initiative
Orlando, FL
robert.chadwick.ctr@adlnet.gov

David Easter
Lockheed Martin
Cary, NC
david.easter@lmco.com

INTRODUCTION

Virtual environments are increasingly used in the training of scientists, engineers and in STEM education (Satyandra K. Gupta, 2008). These powerful tools allow users to collaboratively interact with simulated systems and environments, increasing trainee engagement. However, those deploying a training experience within a virtual environment face several technical challenges that make it difficult or impossible to reach the intended audience. Primarily, these challenges consist of accessibility and network requirements (Freitas, 2008). The Virtual World Framework (VWF) project sponsored by OSD, Training Readiness & Strategy, seeks to lower these barriers to entry by creating a web browser based virtual environment platform that uses a novel network architecture to deliver synchronized, collaborative, multi-user experiences with modest network requirements. This framework enables the authoring of immersive experiences that are available instantly on any hardware that supports a modern web browser - without special network configuration or additional client side software. This enables training to be delivered to users in situations where traditional approaches would not be possible.

DESIGN PRINCIPLES

Open source

A primary goal of the VWF project is to provide an open source platform for future development in the field of collaborative training applications. It is necessary that the tools, knowledge, and technologies created by the project are available to outside developers both present and future. Code is available immediately as contributors modify it, and outside users are free to use and modify the system as they see fit. The VWF project team regularly reviews work by outside developers for suitability as an inclusion in the main VWF source.

Web Based

Traditional virtual environments can be difficult to install and run. Many require custom software to be installed on the client machine, which presents an obstacle to deployment in many enterprises. Several other products are based on Java, which has known security vulnerabilities and entails an unacceptable risk for military organizations. Additionally, various products have version compatibility issues with some operating systems, or cannot be run by mobile OS's. The VWF seeks to build a virtual environment and application architecture that will run on any

hardware configuration and on any operating system, including mobile devices. The system should not require any software to be installed by the client, minimizing barriers to deployment. The VWF is authored as an HTML5 application, and uses several emerging standards supported by modern web browsers.

Client Side Execution

In order to allow the VWF to scale efficiently, the simulations are not computed by a central server. While a server is necessary to deliver content and synchronize the client state, the server does not actively participate in the simulation. This allows a single server to host many applications and serve hundreds of users simultaneously. Additionally, each client is able to present the user with an optimal experience for their device and platform.

Low Bandwidth

The VWF must continue to function in low bandwidth environments. This goal is achieved through a unique approach to simulation synchronization called "Replicated Computation." Each client carries out the simulation independently, but is synchronized by careful management within the VWF Kernel. Network messages need only be moved between clients when an outside event injects input into the simulation. In this case, the input event is sent out to a server which delivers it to all participating clients. Each client then independently computes the identical application state based on the input, thus maintaining state synchronization with minimal bandwidth requirements.

SYSTEM ARCHITECTURE

In order to fulfill the requirement that the VWF system be open source and web based, it is designed as a client side JavaScript library, and utilizes the capabilities of HTML5 browsers. These capabilities allow the VWF to provide rich graphics and networking functions without relying on any proprietary browser plug-in software. Additionally, the VWF contains a small server component consisting of a static file server and a message bus component called the Reflector. These components work together to create a platform that allows applications to be synchronized across multiple clients.

The VWF design implements a Model-View-Controller architecture, with the client side and Reflector components working together to create a distributed model. While this model is actually composed of complex client/server interactions, it is presented to the client side code as a single entity, allowing for network synchronization that is invisible to the application programmer. This makes it possible for an application developer to focus on the business logic within the application, and offload all thoughts about networking to the framework. The internal design of each component is discussed below.

Server Architecture

File Server

This server side component is responsible for two tasks: passing messages between clients, and serving static HTML content. While it would be technically possible to separate the two functions into individual applications or even onto different servers, the current implementation prefers the simplicity of implementing both components within single server side application.

The HTML server component delivers the content required by the simulation to the client application, and also sends the client side application itself to a user who browses to a specific URL. This content includes all the scripts necessary to execute the application, images, sounds, 3D models, textures and other asset files. The server parses the

incoming URL, and determines if that URL references a VWF application. If so, the server serves the VWF bootstrap file. Otherwise, the request is treated as a request for a static file which, if found on the disk, is sent to the client.

Applications and Instances

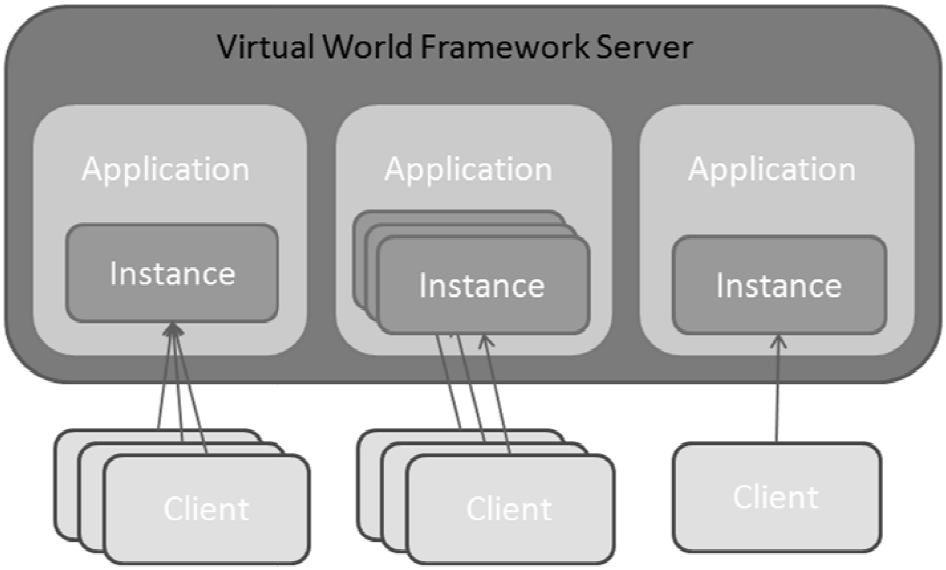

Figure 1. Application Instances

A VWF Application is a program hosted on the VWF file server that uses the VWF architecture to implement its business logic. By implementing the application within the VWF, that application is able to share its state with multiple users, and thus becomes collaborative in nature. Because applications execute on the client, and use the server only for synchronization, the server is able to host many applications simultaneously, even running the same application multiple times (see Figure 1). Each executing copy of an application is called an "Application Instance," and is allocated its own pool of connections on the Reflector.

When a web browser accesses the URL of a VWF application, it fetches and returns to the client an HTML document which causes the client to bootstrap the VWF system. The client now issues requests to the server as instructed by the VWF bootstrap file, fetching system components and building the client side application. When this process is complete, the client connects to the server via a WebSocket, and is then instructed to make a request for the initial application state. This state is parsed and loaded into the client side VWF application, which begins executing the simulation.

Reflector

The final and most important server side responsibility of the VWF application is the synchronization of application instances. This is accomplished by a portion of the server called the Reflector. The primary responsibility of the Reflector is to pass messages between clients, and organize those clients into groups based on the Application Instance. By sorting clients which have all loaded the same initial VWF application state into separate groups, and

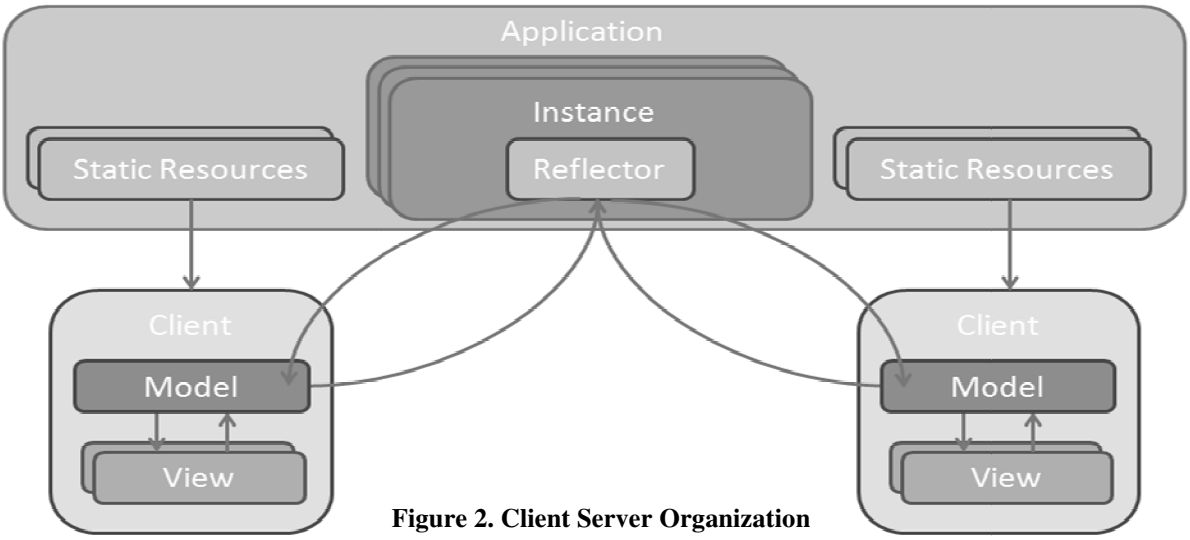

Figure 2. Client Server Organization

only passing messages between group members, the Reflector is able to segment all users such that each group shares a common synchronized copy of an application. Thus, a single VWF application which implements a collaborative document editor could be used simultaneously by many pairs of users, each pair editing their own document.

As will be discussed in the Client Architecture section, the VWF maintains synchronization between clients by carefully imposing rules on how applications may change their internal state. Certain types of state updates (i.e. responding to user input) require a message to be "bounced" off the Reflector server. This bounce is transparent to the programmer, but allows other clients connected to the same Application Instance to be notified of the state change. The Reflector receives these messages and sends them to each client connected to an application instance (see Figure 2). These clients then respond to the event, thus updating their internal state in sync.

Client Architecture

Kernel

The core of the VWF client side application is called the Kernel. This component is the framework in which the other modular components are hosted, and is responsible for routing messages between portions of the system. The Kernel also implements the replicated computation synchronization mechanism by forming a connection to the Reflector server, and routing certain messages through the Reflector. Additionally, the Kernel is responsible for decomposing VWF node definitions into a series of properties, events, methods and behaviors, and issuing commands to the system regarding the creation of these entities. The kernel is the first part of the system loaded by the bootstrap file.

Drivers

Drivers are a modular component of the VWF system which are used to provide capabilities or services to the simulation. Physics, graphics, audio, input, and scripting are each implemented as separate drivers, allowing applications which do not require one of these features to avoid loading capabilities that are superfluous.

The VWF API makes an important distinction between two types of drivers. Model drivers maintain the shared application state, while View drivers provide a mechanism for that state to be presented to the clients in varying yet consistent ways. While Model drivers can directly influence the simulation (i.e., computing object position with a physics engine) and must be loaded by all participants in an Application Instance, View drivers are indepant of the simulation, and my vary per client.

Drivers supply features to a VWF application by responding to messages sent from the Kernel. These messages take the form of events that describe the changing state of the internal application data. For instance, the scripting driver may execute an object script which calls for a sound to be played. The kernel would distribute a "play the sound file at this URL" message to all the drivers, one of which would be programmed to respond to that particular message. This driver (in this case, the sound driver) would respond to the message by loading and playing the sound file. In this example, the driver would be implemented as a View driver, as the sound is user output, and cannot be used to affect the internal state of the simulation.

As another example, consider a driver that supports physics. This driver may respond to a Kernel message that approximately means 'are these two object interpenetrating?' The result of this Kernel message would be passed back to the script driver, which then may make some branching decision based on the result. Because the state of the simulation depends on the result of the drivers' operation, the physics driver must be implemented as a Model driver.

Nodes

Each entity, object, or actor within a VWF simulation is referred to as a node. Nodes are the basis for the business logic of a simulation, and are carefully managed by the VWF to ensure synchronized internal state. The VWF provides a rich set of structures to allow for the reuse of nodes, and for allowing nodes to interact with each other. Nodes are arranged hierarchically, and roughly correspond to a scene graph from traditional computer graphics. Nodes are defined by the properties they contain, the methods they may execute, and the definition of each of their children.

The simulation actually occurs as events injected into the system cause nodes to change their internal state. These inputs can be user inputs (as injected by a view driver) or systems inputs such as initialization notifications, or "ticks." 'Tick" messages allow nodes to update their state over time. Additionally, models may generate events based on the properties of the nodes in the simulation, such as triggering a "collision" event when a node changes its position such that it penetrates another node's physical representation. View drivers are notified by the kernel about the changing properties of the simulation, and respond by presenting output to the user.

The VWF provides structures to make it easier to author the interrelations between nodes that create the simulation. Application code may traverse the hierarchy of nodes that are taking part in the simulation, and modify the properties therein. They may also search for specific other nodes, and bind code blocks to be triggered when some event occurs on that node. They may even add or remove child nodes from themselves or from others. While all this interaction occurs, the kernel is constantly communicating with the drivers, which may choose to inject events, or may display the changing state. All of these features are contained within the VWF API, which safeguards the simulation from drifting out of synchronization. When the kernel detects some operation that will cause replicated computation to fail, and thus produce different results on each client, it forwards the event through the Reflector before passing it on to the local nodes or drivers, thus ensuring that each client remains synced.

Components

Because one of the key goals of the VWF is to provide for reusable objects that can contain their own logic and functions, nodes must be able to reuse functionality defined previously. The VWF supports this two ways.

First, a node may be created within the system that extends and reuses some stored node definition. These stored definitions are termed Components. Any node can become a Component by being saved out into the File Server in a particular format. Future nodes can base their definition on the Component to extend, modify, or reuse its logic. This mechanism allows a library of objects to be created that can be reused multiple times within an application, or within multiple applications. Components can also extend other components, providing a rich inheritance structure for building complex objects out of simple ones. Many of the basic object types within the system are implemented as components, including scenes, camera, and particle systems.

Additionally, a node component can implement one or more "behaviors." A behavior is simply a set of properties, methods and events that are added to a node from an external file. Behaviors allow the simulation author to create reusable functions that can be added to a node, regardless of that node's inheritance chain.

Out of the box, the VWF provides a set of node components that implement a useful API for loading and manipulating models in 3D space. These components include cameras, particle systems, physical objects, lights and points. Through the inheritance model, the author can extend these components into more specific objects. For instance, the simulation author may create a new component based on the particle system component. This new component may modify the properties of the underlying particle system component to create a fire effect, and expose new properties such as "FireTemperature" or "BurnLength." Modifying these new properties could be programmed to change the underlying particle system to create the effect of a bluer or yellower fire, or a higher flame. Finally, the author could create several nodes based on their fire component to display a scene with multiple fires of various sizes and colors.

SYSTEM REQUIREMENTS

The choice to base the VWF on HTML5 imposes some system requirements on the client machine. In order for a browser to connect to the VWF server and execute a VWF simulation, it must support at minimum ECMAScript5, and WebSockets. These capabilities are APIs implemented in most modern browsers and standardized by industry. Any browser that supports these standards can participate in a VWF environment. This includes several current mobile browsers.

Client Side Requirements

WebGL

WebGL is a JavaScript API that exposes the same functionality as OpenGL ES 2.0. It allows a browser to command the system GPU from the script located within an HTML document, and can display the results of the GPU operations within a section of that document. This allows a web browser to render three dimensional environments or objects, or possibly to accelerate other types of calculations by leveraging the computation power of a GPU. The default functionality of the VWF application is to use this technology to display the visual output of all simulations. Currently, 72.6 percent of desktop browsers and 9.6 percent of mobile users can support WebGL. While the VWF is able to execute in a mode that does not require WebGL, 3D immersive VWF applications do require this technology.

ECMAScript 5

ECMAScript is a standardization of the common JavaScript web scripting language. All VWF functionality, both within the VWF drivers and kernel, and within the simulation logic itself, is implemented in ECMAScript 5. In order

to provide the highest quality code, the VWF makes extensive use of the new features within version 5, including getters, setters, and "strict mode."

WebSockets

Because the VWF client side application must be able to receive commands from the remote server, it establishes a WebSocket connection to the Reflector. A WebSocket is a full-duplex communication channel between a web server and a client based on a TCP connection, via a protocol that was standardized by the IETF as RFC 6455 in 2011. WebSockets allow for high speed links between servers and clients that can remain open for the duration of a users visit to a webpage. This differs from previous technologies, which would constantly create new temporary connections to the server to poll for updates. WebSockets allow the VWF to receive simulation messages from the Reflector server far faster and with less overhead than traditional HTTP techniques. Additionally, a WebSocket connection can be established over port 80 simultaneously with the regular HTTP web traffic, making it possible to form the connection even from behind most firewalls. Currently, more than 70% of browsers support this requirement. (Deveria, WebGL - 3D Canvas graphics, 2013)

Server Side Requirements

Because of the low impact server design mentioned previously, the server component of the VWF has very few requirements. The server software must implement the WebSocket protocol, and serve static HTML content, but otherwise has no particular requirements. It should be noted that, as with any server software, the VWF server component does require access through any firewalls on at least one port.

CHALLENGES AND CONSIDERATIONS

Platform Support

Our team has tested the VWF client application on a range of browsers and system configurations, and largely met with success. The current implementation has been shown to function under Windows XP, Windows Vista, Windows 7, Windows 8, OSX, Android, and Linux. Browser support varies over time as the underlying specifications are still in flux, but generally, we find the system is well supported in Chrome, Firefox, Safari, and Opera 15. Current versions of Internet Explorer may use some VWF applications, but compatibility is limited by the lack of WebGL support. This significant impediment may be alleviated in the future, as there is evidence that Internet Explorer 11 may contain WebGL support. We also expect to see increasing support in the mobile space, as browser vendors continue to implement mobile versions of these cutting edge features.

Client Side ECMAscript Performance

One of the foremost concerns we faced when considering the feasibility of an HTML5 client side virtual environment was the slow performance of JavaScript execution. It's well understood that the raw performance of code is generally less than that of native code platforms like C++. However, our claim that the web platform would mature sufficiently to allow for complex applications in the browser was vindicated, and we no longer find client side performance to be a bottle neck. While there are some applications where raw compute performance can still be an issue (physics, collision), browser vendors continue to improve on JavaScript speed. Currently, there are several projects to add faster computation the web platform, both by modifying the JavaScript standard, and by leveraging GPU hardware. WebCL, an open project from the Kronos Group, may eventually enable web applications to harness GPU and multi-core CPU parallel processing, enabling significant acceleration of applications such as advanced

physics. Additionally, the Mozilla Foundation is working to standardize a subset of the JavaScript language in a system called ASM.js. ASM.js may make it possible to compile large JavaScript applications to an intermediate code that, when paired with specific optimizations within browsers, can achieve near native performance. As these and other innovations become available, VWF will evolve to take advantage of their performance benefits. It should be noted that mobile devices have significantly lower performance profiles, and on these devices script performance does hinder some VWF usability.

Real Time Synchronization

The VWF system guarantees state synchronization at the simulation time step interval, but not at real time intervals. The Replicated Computation principle requires each client to execute the entire simulation at each time step, and does not allow one client to 'cheat' to catch up to real time. Thus, if one client takes longer than one real second to simulate one second of simulation time, that client simply cannot interact with a human user in real time. It should be noted that this client will still always remain in sync relative to the simulation timestamp – it will simply display the results of the simulation several seconds or minutes after the faster client. Obviously, this limits the ability of a human to interact. In practice, simulations should be no more complex than can be simulated in real time by the slowest client.

Difficulties Authoring Simulation Scripts

The "Replicated Computation" design, which allows the VWF to operate with very low bandwidth and very little impact on the server, also introduces some complexities to the simulation logic author. Namely, state must neither be stored in local JavaScript variables, nor can the scripts that drive the simulation directly accept input from an outside source. This limits the ability of the system to integrate with outside services, as they may introduce inconsistencies in the simulation. Imagine for instance the following scenario – a virtual lobby contains an artwork that changes its pose based on the information in a data feed of DJIA stock market prices. When the simulation updates, a request for the current DJIA data is sent to an outside server that returns the instantaneous value of the index at that moment. Because the simulation update cycle may run many milliseconds behind on one vs. another, but stock market data is keyed to real world clock time, the pose of the statue at a given second in simulation time may differ. Should a bullet intersect with the artwork on one client, it would not be guaranteed to intersect it on the other, leading to cascading failures of synchronization.

Currently, the VWF has few mechanisms to prevent this sort of problem. While we do provide some tools to avoid this, such as a synchronized random number generator, there remain many possibilities for error. Best practices documentation should help the programmer avoid asynchronization of client scripts, but these recommendations cannot be enforced by the system. Simulation authors must be aware of the nature of replicated computation, and avoid introducing dependencies on outside data into their simulations. We are actively researching ways to sanitize client scripts to avoid these problems, but it remains very much an area for research.

FUTURE WORK

Driver Configuration

Use of a modular system of drivers allows the VWF to avoid imposing policy on its applications while allowing a great deal of flexibility in the types of applications that can be developed. Currently, applications must provide a configuration file that specifies the drivers necessary to support the components that the application uses. In most cases, this is redundant since it would be possible for drivers to declare the components that they support. With a

registry collecting this data, the VWF could automatically locate and load drivers as needed. Configuration files would still be available for custom configurations. Future work will investigate ways for drivers to be loaded and unloaded during the execution of an application, and for driver state to be made current with the application without requiring the developer to write detailed and error-prone synchronization algorithms.

View Components

VWF components allow applications to control the elements of a simulation while still having a measure of isolation from the underlying implementation. A component that performs a characteristic animation, for example, can successfully control an animation on any node, whether it is a keyframe animation imported from a Collada document, or a custom rendering scripted directly on the node. Components may directly read and write properties, invoke methods, and respond to events while the system ensures that the actions implied are correctly executed on each client.

However, this only applies to the shared state--the "model" portion of the application. Parts that require access to the user or to events that are not available to all clients--the "view" portion--are provided much less support. They must connect to the VWF kernel through its view API, issue actions indirectly using object IDs, and carefully match actions to results when return values are required. A direct corollary to model components that is adapted to the special characteristics of the application's view side is needed. View components would allow functions that manipulate the simulation but require access to the user to be developed more easily.

Additionally, many functions such as UI widgets or editing annotations operate exclusively on one client at a time and don't affect the shared state. These functions would benefit from local model-like access to the same systems that the global model and view components manipulate. Using the same API as model components, and in some cases using model components directly, local components could create direct and private interactions with the user while maintaining a degree of isolation from the underlying systems.

Modular Drivers

Many opportunities to extend VWF functionality forgo the isolation of the component API for the greater fidelity allowed by directly manipulating a driver's internal data. Currently, this is only possible if all of the functions that have access to this data are part of the same driver. To allow extensions such as specialized 3-D rendering techniques to be easily supported, a lower-level extension system is needed. Such a system would provide a way for cooperating drivers to conditionally share internal objects, such as specific nodes from a particular 3-D scene manager. Allowing drivers to support only certain objects or certain types would let the system perform more of the work and reduce the work required by the driver developer.

WebRTC Integration

A useful effect of the network model implemented by the VWF is the ability for users to collaborate on tasks. This collaboration requires communication between users so that they may coordinate their efforts. Currently, the VWF implements some example of text based chat clients, but we feel that face-to-face vocal communication is a richer medium for interaction. The VWF recently added a video chat service based on emerging W3C standard called WebRTC. WebRTC makes it possible to form peer-to-peer video and audio communication channels between users within the same VWF application instance. We expect this will greatly increase the utility of the platform, without

violating the central design principal of client side computation. Notably, this approach will not necessitate that the video and audio content be routed through the VWF server, maintaining the scalability of the server side application.

Reflector Security

Currently, all clients connected to a VWF application instance are treated as peers. This creates some problems in situations where it is desirable that some users be able to experience an application without the ability to modify the internals of the simulation. Future work will include adding a security model to the Reflector, which will enforce some policy requirements on incoming messages. Clients will have to claim ownership of a WebSocket connection, and supply credentials that will be validated against a database. The form of that database is yet to be determined, and it may be prudent to engineer the system in such a way that it is agnostic as to the method of credential validation. Regardless, once validated, the system will apply the appropriate security to the client by filtering its inputs to the simulation, and refusing to pass on messages that violate the policy. In this manor, a client will be afforded the permission to edit the state of some objects but not others. While the client can always change the state of their own internal simulation, the Reflector will prevent a malicious user from effecting the simulation on peer machines.

CONCLUSION

Our work on the Virtual World Framework system has convinced us that building a virtual world training application on the HTML5 platform is a viable way to lower the barriers to entry in the world of immersive simulation. Much like the advent of webmail, bringing 3D environments into the browser makes them more accessible and broadens the base of potential users. A web-based design makes it possible to deliver content on multiple devices and platforms, while authoring only one set of client-side code. Additionally, moving to the web platform makes it possible to integrate simulation with existing systems such as learning management systems, massively online open courses, and content repositories. The practicality of designing a client-side simulation environment introduces interesting challenges, necessitating tradeoffs in several areas. However, we believe our work so far makes a compelling case that the benefits of this approach outweigh the cost.

REFERENCES

Chris Dede, J. C. (2005). *"Students' Motivation and Learning of Science in a Multi-User VirtualEnvironment"*.

Deveria, A. (2013). *WebGL - 3D Canvas graphics*. Retrieved from Can I use...: http://caniuse.com/#feat=webgl

Deveria, A. (2013). *Web Sockets*. Retrieved from Can I use...: http://caniuse.com/#feat=websockets

Freitas, S. d. (2008). *Serious Virtual Worlds A scoping study.* Serious Graphics Institute.

Satyandra K. Gupta, D. K. (2008). *Training in Virtual Environments.* College Park, Maryland: CALCE EPSC Press CECD/ETC Series.

Boyd, Richard. (2012) "*The Virtual World Ecosystem Framework"*